Proof Plans for the Correction of False Conjectures

Proof Plans for the Correction of False Conjectures *

Raul Monroy, Alan Bundy, & Andrew Ireland

Department of Artificial Intelligence
The University of Edinburgh
80 South Bridge, EH1 1HN
Scotland, U.K
raulm, bundy, & air@aisb.ed.ac.uk

Abstract. Theorem proving is the systematic derivation of a mathematical proof from a set of axioms by the use of rules of inference. We are interested in a related but far less explored problem: the analysis and correction of false conjectures, especially where that correction involves finding a collection of antecedents that, together with a set of axioms, transform non-theorems into theorems. Most failed search trees are huge, and special care is to be taken in order to tackle the combinatorial explosion phenomenon. Fortunately, the planning search space generated by proof plans, see [1], are moderately small. We have explored the possibility of using this technique in the implementation of an abduction mechanism to correct non-theorems.

1 Introduction

The problem of building an artificial mathematician to find a mathematical proof has been a topic of much interest in Artificial Intelligence. We are interested in a related but far less explored problem: the analysis and correction of false conjectures, especially where that correction involves finding a collection of antecedents that, together with a set of axioms, transform non-theorems into theorems. More formally, and following [5]:

Given a set of axioms $\mathcal{A}$ and a false conjecture G, i.e. $\mathcal{A} \rightarrow G$ does not hold, our aim is to identify C such that:

1. $\mathcal{A} \wedge C \rightarrow G$ is a theorem, i.e. the addition of C turns the non-theorem into a theorem;
2. $\mathcal{A} \wedge C$ is satisfiable, i.e. C is *consistent* with the set of axioms;
3. $C \rightarrow G$ does not hold, i.e. C is *nontrivial*; and
4. C is *minimal* in that it does not contain any redundant literals.

* We are grateful to Jane Hesketh and the anonymous referees for their useful comments on an earlier draft of this paper. The research reported here was supported by SERC grant GR/H/23610 to the second and third author, and ITESM & CONACyT studentship 64745 to the first author.

By way of motivation, consider the following non-theorem [2]

$$\forall N : \text{nat. } \text{double}(\text{half}(N)) = N \tag{1}$$

where the functions double and half have their natural interpretation returning twice and half their inputs, respectively. Clearly, a condition like $N < 0$ does not meet our requirements because it is inconsistent with sort/type information. In addition, the formula

$$\forall N : \text{nat. } (\text{double}(\text{half}(N)) = N) \rightarrow (\text{double}(\text{half}(N)) = N)$$

is not a useful solution since the condition is trivial. The abduction mechanism we present in this paper is capable of finding the condition $\text{even}(N)$, which is clearly consistent, nontrivial, and minimal. Note that a condition of the form $\text{even}(N) \wedge N \neq \text{s}(0)$ would not be minimal because the second conjunct follows from the definition of the predicate even.

2 Proof Plans

Reasoning and searching are necessary for the solution to the problem of correcting a false conjecture. *Abduction* seems to be a candidate mechanism for the former. Abduction, as proposed by C.S. Peirce [13], is a fundamental form of logical inference that allows us to find hypotheses that account for some observed facts. Its simplest form is:

From A $\rightarrow$ B, and B
Infer A as a possible justification of B

Most of the mechanisms for driving the generation of abductive hypotheses are based on resolution (see [12] or [11] for a survey on abduction mechanisms). However, most failed proof search spaces are huge and these mechanisms are severely affected by the combinatorial explosion phenomenon, see [16].

Fortunately, the planning search spaces generated by *proof plans* are moderately small, see [1]. This technique guides the search for a proof in the context of tactical style reasoning [8]. Tactic specifications called *methods* express the preconditions under which a tactic is applicable and the effects of applying such a tactic. The proof plan technique has been implemented in a system called CLAM [3] and successfully applied to the domain of inductive proofs [2]. In this paper, we show how to implement an abduction mechanism using plans for inductive proofs. The mechanism relies on the meta-level reasoning used for forming a proof plan, since it provides a basis for analysing failed proof attempts.

[2] Following the Prolog convention, we denote variables with symbols that start with an upper-case letter.

2.1 Rippling

The key idea behind inductive proofs is the use of induction hypotheses in completing step-case proof obligations. The search control heuristic called *rippling* [4] was designed for this task. It works by applying a special syntactic class of rewrite rules called *wave-rules*. The simplest form of such a wave-rule gives rise to the following schema:

$$F(\boxed{S(\underline{U})}^{\uparrow}) \Rightarrow \boxed{T(F(\underline{U}))}^{\uparrow} \tag{2}$$

where F, S, and T are functors. Note that T may be empty while S and F may not. F and $\boxed{S(\underline{U})}^{\uparrow}$ are called *wave-function* and *wave-term*, respectively. Wave-terms are composed of a *wave-front* and one or more *wave-holes*. Wave-holes are the underlined sub-terms of wave-terms. Sub-expressions of the induction conclusion that also appear in the hypothesis are either underlined or not enclosed by boxes. For our current wave-rule example, F and U would match such sub-expressions. Note how the application of (2) has the effect of moving the S through the F. Also, note that the arrow indicates the direction in which wave-fronts are moved within the term structure.

By marking these wave-terms and tracking their movements, we can ensure that our rewriting makes progress towards the desired effect: the removal of the obstructive wave-fronts so that *fertilization* can be applied. Fertilization, according to Boyer and Moore, is the process of applying an induction hypothesis.

2.2 Proof Critics

Experience has shown that a failed proof attempt may hold the key for discovering a complete proof. In [9], the author proposes the use of *planning critics* as a mechanism to provide the means of exploiting failure and partial success in the search for a proof. Planning critics are aimed at capturing our intuition as to how a partial proof can be completed. For this reason, proof critics are associated with proof methods. Any time the application of a particular proof method fails, a collection (possibly empty) of planning critics is invoked. Their application often results in a modification of either the current plan structure, the given conjecture, or the theory in which we are working.

3 Correcting Faulty Conjectures

Our abduction mechanism to correct faulty theorems is built upon CLAM. It consists of a collection of proof critics that define heuristics to detect, isolate, and correct some kinds of faults. Generally speaking, the mechanism works as follows. Let us assume we are given a conjecture, say G. We first let CLAM attempt to find an inductive proof plan for G. If the conjecture is faulty, this process will fail and terminate pointing at an unprovable sub-goal that arose from one case of the inductive proof. According to the point at which failure

occurred (c.f. proof methods), a particular collection of critics is then invoked to perform a syntactic analysis on the unprovable sub-goal. From such an analysis, we build the condition that is to be added to the current conjecture. Often, these unprovable sub-goals represent contradictions to either the current set of axioms or sort/type information.

False conjectures that exhibit faults in boundary values were successfully corrected using the information provided by the base case proof obligation. We worked by refinement when a suggested condition from a previous patching attempt turned out to be necessary but not sufficient. We also corrected false conjectures in which the fault exhibited arguments in wrong positions within the conjecture structure; this sort of fault can be found in attempts at proving commutativity of operators that are not Abelian. In the following sections, we introduce the definition of some proof critics of the abduction mechanism by example.

3.1 Exploiting Contradictory Blocked Goals

Consider the non-theorem:

$$\forall A, B : \text{list}(DataType).\ \text{length}(A <> B) > \text{length}(A) \tag{3}$$

The recursive definitions of <>, >, and length give rise to the rewrite rules[3]:

$$\boxed{X :: \underline{U}}^{\uparrow} <> V \Rightarrow \boxed{X :: \underline{U <> V}}^{\uparrow} \tag{4}$$

$$\text{nil} <> U \Rightarrow U$$

$$\boxed{\text{s}(\underline{X})}^{\uparrow} > \boxed{\text{s}(\underline{Y})}^{\uparrow} \Rightarrow X > Y \tag{5}$$

$$X > 0 \Rightarrow X \neq 0$$

$$0 > X \Rightarrow \text{false}$$

$$\text{length}(\boxed{X :: \underline{U}}^{\uparrow}) \Rightarrow \boxed{\text{s}(\underline{\text{length}(U)})}^{\uparrow} \tag{6}$$

$$\text{length}(\text{nil}) \Rightarrow 0$$

We attempt to prove (3) using the primitive induction on lists selecting A as the induction variable[4]. The base case ($A = \text{nil}$) leads to the following sub-goal

$$\forall B : \text{list}(DataType).\ \text{length}(B) \neq 0 \tag{7}$$

With (7), a nested induction is suggested, $v_n :: B$. This time the base case ($B = \text{nil}$) gives rise to a contradictory blocked goal:

$$\text{length}(\text{nil}) \neq 0$$

$$0 \neq 0$$

[3] The operators ::, <>, and s() represent the infix list constructor function, the lists concatenation function, and the successor constructor function, respectively.

[4] This will be abbreviated as $IndScheme[IndVar]$; where $IndVar$ is the induction variable, and $IndScheme$ is the suggested induction rule of inference.

Definition 1 Contradictory Blocked Goals. A goal G is said to be *contradictory blocked* if it cannot be further rewritten, all its variables are instantiated, and it is false in the domain of the theory in which we are working.

This contradiction suggests our first patch, namely, to introduce $B \neq \text{nil}$, i.e. the negation of the base case for the most recent induction, as a condition to the original conjecture. Note that by omitting this case condition, our method guarantees that the contradictory blocked goal will not be experienced again. Hence, we have a new conjecture of the form:

$$\forall A, B : \text{list}(DataType).\ B \neq \text{nil} \rightarrow \text{length}(A <> B) > \text{length}(A) \qquad (8)$$

With the revised conjecture (8), a $v_n :: A$ induction schema is again suggested. This time the base case proof obligation goes through and so does the step case. In the step case we have an induction hypothesis of the form:

$$\forall B : \text{list}(DataType).\ B \neq \text{nil} \rightarrow \text{length}(a <> B) > \text{length}(a) \qquad (9)$$

and an initial induction conclusion of the form:

$$b \neq \text{nil} \rightarrow \text{length}(\boxed{v_0 :: \underline{a}}^{\uparrow} <> b) > \text{length}(\boxed{v_0 :: \underline{a}}^{\uparrow}) \qquad (10)$$

Rippling-out (10) with (4) results in:

$$b \neq \text{nil} \rightarrow \text{length}(\boxed{v_0 :: \underline{a <> b}}^{\uparrow}) > \text{length}(\boxed{v_0 :: \underline{a}}^{\uparrow})$$

By wave-rule (6) this rewrites both, the right-hand side (RHS), and the left-hand side (LHS) of the above formula to give us:

$$b \neq \text{nil} \rightarrow \boxed{s(\underline{\text{length}(a <> b)})}^{\uparrow} > \boxed{s(\underline{\text{length}(a)})}^{\uparrow}$$

and finally, wave-rule (5) gives us:

$$b \neq \text{nil} \rightarrow \text{length}(a <> b) > \text{length}(a)$$

Note that this expression matches the induction hypothesis (9). We can appeal therefore directly to the hypothesis to complete the proof. This process is called *strong fertilization.*

The critic definition depicted in Fig. 1 provides a general explanation of the mechanism.

CRITIC induction

Input:	$Plan$,	; If current goal G is
	node($Plan, [], [] \vdash Goal$),	; contradictory blocked,
	current_node($Plan, Address, H \vdash G$)	; negate the condition
Precondition:	contradictory_blocked_goal(G)	; for the most recent
Patch:	failed_at($Plan, Address, Case$),	; induction and add it
	insert_condition($\neg Case, Goal, NewGoal$),	; as a condition to the
	resume_plan($Plan, [], [] \vdash NewGoal$)	; original goal $Goal$

Meanings of the meta-logic terms:

- node($Plan, Address, Sequent$) is used to access the sequent recorded at node $Address$. [] denotes the root node.
- current_node($Plan, Address, Seq$) is used to get the address of the current node, and to access the sequent recorded at that node.
- failed_at($Plan, Address, Case$) means that $Case$ is the case at which failure occurred in the most recent induction.
- insert_condition($Cond, F, NewF$) means $NewF$ is the result of inserting condition $Cond$ in conjecture F.
- resume_plan($Plan, Address, Sequent$) resumes the proof plan formation of $Plan$.

Fig. 1. Exploiting contradictory blocked goals

3.2 On Fixing Non-Theorems by Refinement

As the reader may now suspect, it is possible to have a false conjecture in which the patch suggested by the above heuristic is not sufficient to transform the non-theorem into a theorem. This situation is likely to occur whenever the condition consists of either a predicate other than equality or a combination of predicates.

As a solution to this problem, we have defined a strategy which supports the refinement of a previous patch. As will become clear later, our strategy exploits both syntactic (rippling) and semantic information. Consider again (1), the example conjecture introduced in Sect. 1. The recursive definitions of double and half give rise to the following rewrites:

$$\begin{aligned} \mathrm{double}(0) &\Rightarrow 0 \\ \mathrm{double}(\boxed{\mathrm{s}(\underline{X})}^{\uparrow}) &\Rightarrow \boxed{\mathrm{s}(\mathrm{s}(\underline{\mathrm{double}(X)}))}^{\uparrow} \qquad (11) \\ \mathrm{half}(0) &\Rightarrow 0 \\ \mathrm{half}(\mathrm{s}(0)) &\Rightarrow 0 \\ \mathrm{half}(\boxed{\mathrm{s}(\mathrm{s}(\underline{X}))}^{\uparrow}) &\Rightarrow \boxed{\mathrm{s}(\underline{\mathrm{half}(X)})}^{\uparrow} \qquad (12) \end{aligned}$$

In addition, we assume that our theory of natural numbers includes the predic-

ates even and odd[5]:

$$\text{even}(0) \Rightarrow \text{true}$$
$$\text{even}(\text{s}(0)) \Rightarrow \text{false} \tag{13}$$
$$\text{even}(\boxed{\text{s}(\text{s}(\underline{X}))}^{\uparrow}) \Rightarrow \text{even}(X) \tag{14}$$
$$\text{odd}(0) \Rightarrow \text{false}$$
$$\text{odd}(\text{s}(0)) \Rightarrow \text{true} \tag{15}$$
$$\text{odd}(\boxed{\text{s}(\text{s}(\underline{X}))}^{\uparrow}) \Rightarrow \text{odd}(X) \tag{16}$$

Furthermore, we assume the wave-rule for the cancellation of the successor function:

$$\boxed{\text{s}(\underline{X})}^{\uparrow} = \boxed{\text{s}(\underline{Y})}^{\uparrow} \Rightarrow X = Y \tag{17}$$

We attempt to prove (1) using $\text{s}(\text{s}(n))$ induction. The first base case ($N = 0$) is trivial. It is the second base case ($N = \text{s}(0)$) which is interesting since it gives rise to a contradiction, as shown below.

$$\text{double}(\text{half}(\text{s}(0))) = \text{s}(0)$$
$$\text{double}(0) = \text{s}(0)$$
$$0 = \text{s}(0)$$

This suggests our first patch attempt of introducing the condition $N \neq \text{s}(0)$ using the strategy defined in the previous section. This gives a new conjecture of the form:

$$\forall N : \text{nat}.\ N \neq \text{s}(0) \rightarrow \text{double}(\text{half}(N)) = N \tag{18}$$

With the revised conjecture, (18), a two step induction is again suggested. This time both base cases go through. In the step case our induction hypothesis is:

$$n \neq \text{s}(0) \rightarrow \text{double}(\text{half}(n)) = n \tag{19}$$

and the initial induction conclusion takes the form:

$$\boxed{\text{s}(\text{s}(\underline{n}))}^{\uparrow} \neq \text{s}(0) \rightarrow \text{double}(\text{half}(\boxed{\text{s}(\text{s}(\underline{n}))}^{\uparrow})) = \boxed{\text{s}(\text{s}(\underline{n}))}^{\uparrow}$$

Rippling-out this formula with (12), (11), and (17) results in:

$$\boxed{\text{s}(\text{s}(\underline{n}))}^{\uparrow} \neq \text{s}(0) \rightarrow \text{double}(\text{half}(n)) = n$$

At this point, any further rippling is blocked. Note how this formula matches the induction hypothesis (19) *modulo* the antecedent. Although strong fertilization is not possible we are *potentially* in a position to perform what is defined as *conditional fertilization*. Conditional fertilization extends strong fertilization

[5] The predicate odd is not needed, but is included to show that the technique does

METHOD conditional_fertilize

Input:	$H \vdash C_{IC} \rightarrow G_{IC}$,	; Current Sequent.
	$\mathrm{hyp}(H, C_{IH} \rightarrow G_{IH})$	; Induction hypothesis.
Preconditions:	$\mathrm{exp\_at}(G_{IC}, Posn) = G_{IH}$,	; Matching modulo antecedent.
	$\mathrm{tautology}(H <> C_{IC} \vdash C_{IH})$	; The condition of the hypothesis
		; is provable given what is known.

Meanings of the meta-logic terms:

- $\mathrm{hyp}(H, Hyp)$ means Hyp is in hypothesis list H.
- $\mathrm{exp\_at}(Exp, Posn)$ is the subexpression in Exp at position $Posn$.
- $\mathrm{tautology}(H \vdash C)$ is true when the condition C is provable given the hypothesis list H.

Fig. 2. Preconditions of the conditional fertilization method

with conditional equations. The preconditions to apply conditional fertilization are shown in Fig. 2.

For our example the first precondition holds while the second is obviously false. The failure of the fertilize method suggests that our initial condition, $N \neq \mathrm{s}(0)$, was *necessary* but not *sufficient* in order to make (1) into a theorem.

Our second attempt at patching (1) is syntactically driven and represents a refinement of our first patch. We analyse the second failure with the aim of finding a wave-function which will not lead to the blockage experienced in the second proof attempt, i.e.

$$\underbrace{\boxed{\mathrm{s}(\mathrm{s}(\underline{n}))}^{\uparrow} \neq \mathrm{s}(0)}_{\text{blockage}} \rightarrow \ldots$$

We are looking for a wave-rule of the form $F(\boxed{\mathrm{s}(\mathrm{s}(\underline{X}))}^{\uparrow}) \Rightarrow \ldots$, since it allows further rippling. In addition, we know that F must be of type nat→bool. Taking these constraints into consideration there are two[6] candidate wave-rules within our theory: (14) and (16). For our current example therefore F may be $\lambda x.\mathrm{even}(x)$ or $\lambda x.\mathrm{odd}(x)$.

Now we exploit our semantic knowledge. From the first patch attempt we know that[7] $F(\mathrm{s}(0))$ must evaluate to false. Looking at rewrites (13) and (15) we see that even is the correct instantiation for F. The corrected conjecture becomes:

$$\forall N : \mathrm{nat}.\ \mathrm{even}(N) \rightarrow \mathrm{double}(\mathrm{half}(N)) = N$$

[6] Note that wave-rule (12) is ruled-out for type reasons.

[7] This ensures that the second attempt at patching (1) subsumes the first one.

which is actually provable.

This strategy is captured in the critic definition given in Fig. 3.

CRITIC conditional_fertilize

Input:	$Plan$,	; Current plan,
	$\text{node}(Plan, [], [] \vdash Goal)$,	; node, and sequent.
	$\text{current\_node}(Plan, Address, H \vdash C_{IC} \rightarrow G_{IC})$,	
	$\text{hyp}(H, C_{IH} \rightarrow G_{IH})$,	
Preconditions:	$\text{exp\_at}(G_{IC}, Posn) = G_{IH}$,	; Syntacticly and
	$\text{context}(Plan, Address, IndVar, CondList)$,	; semanticly-guided
	$\text{exp\_at}(C_{IC}, Blockage)$,	; partial wave-rule
	$\text{match\_wave\_rule}(Blockage, CondList, F)$,	; matching.
Patch:	$\text{subsumption\_checking}(F(IndVar), CondList, NewCondList)$,	
	$\text{insert\_condList}(NewCondList, Goal, NewGoal)$,	
	$\text{resume\_plan}(Plan, [], [] \vdash NewGoal)$	

Meaning of the meta-logical terms:

- $\text{context}(Plan, Address, IndVar, CondList)$ is used to access the variable which is being inducted upon, and the current set of abducted conditions.
- $\text{match\_wave\_rule}(Blockage, Conditions, F)$ means F is the main functor of a wave-rule whose LHS matches $Blockage$ and the definition of F is consistent with $Conditions$.
  $\text{subsumption\_checking}(P, CondList, NewCL)$ means $NewCL$ is as $CondList$ except that the conditions subsumed by the definition of P have been removed.
- $\text{insert\_condList}(List, G, NewG)$ inserts each condition of $List$ in G.

Fig. 3. Refining previous patching attempts

3.3 Lochs and Dykes

Consider the following faulty conjecture:

$$\forall A, B : \text{list}(T).\ \text{rev}(\text{rev}(A <> B)) = \text{rev}(\text{rev}(B)) <> \text{rev}(\text{rev}(A)) \qquad (20)$$

This formula is false in that the RHS has two arguments in wrong positions. We assume the rewrite rules derived from the definition of $<>$ and the following rewrite rules:

$$\text{rev}(\boxed{X :: \underline{U}}^{\uparrow}) \Rightarrow \boxed{\underline{\text{rev}(U)} <> X :: \text{nil}}^{\uparrow} \qquad (21)$$

$$\text{rev}(\text{nil}) \Rightarrow \text{nil}$$

$$\text{rev}(\boxed{\underline{U} <> X :: \text{nil}}^{\uparrow}) \Rightarrow \boxed{X :: \underline{\text{rev}(U)}}^{\uparrow} \qquad (22)$$

$$\boxed{X :: \underline{U}}^{\uparrow} = \boxed{X :: \underline{V}}^{\uparrow} \Rightarrow U = V \tag{23}$$

We attempt to prove (20) using a $v_n :: A$ induction. The base case is trivial. The step case proceeds as follows. Our induction hypothesis is the following:

$$\forall B : \text{list}(T).\ \text{rev}(\text{rev}(a <> B)) = \text{rev}(\text{rev}(B)) <> \text{rev}(\text{rev}(a)) \tag{24}$$

and the initial induction conclusion is:

$$\text{rev}(\text{rev}(\boxed{v_1 :: \underline{a}}^{\uparrow} <> b)) = \text{rev}(\text{rev}(b)) <> \text{rev}(\text{rev}(\boxed{v_1 :: \underline{a}}^{\uparrow}))$$

By applying wave-rules (4), (21), and (22), we get

$$\boxed{v_1 :: \underline{\text{rev}(\text{rev}(a <> b))}}^{\uparrow} = \text{rev}(\text{rev}(b)) <> \boxed{v_1 :: \underline{\text{rev}(\text{rev}(a))}}^{\uparrow} \tag{25}$$

At this point, no further rewriting is possible but *weak fertilization* is applicable. The use of the induction hypothesis as a rewrite rule is called weak fertilization. Having fertilized (25), the resulting formula is considered as a sub-goal to be proved using (a nested) induction. However, any proof attempt will be fruitless because the conjecture is false. The problem is that we cannot assume this in advance. As a partial solution we have implemented a simple counter-example finder that evaluates a few standard instantiations to check whether a given formula is trivially unprovable. The counter-example finder provides us with the means of detecting a faulty occurrence.

It is clear that the LHS of (25) is fully rippled, whereas its RHS is blocked. According to the rippling paradigm, we say that the wave-fronts on the RHS cannot ripple-out all the way up to the very top of that side. We may think that there is a *dyke*, i.e. a barrier, in the middle of the loch such that it is not possible for the waves to raise up in the conjecture structure.

Our failure location process is guided by the partial use of the induction hypothesis. This process is called *lemma calculation* [10] and is simply the implementation of weak fertilization as a proof critic. It is invoked whenever rippling gets blocked and there exists the opportunity to partially exploit the induction hypothesis.

For our example, the lemma calculation technique would first apply the induction hypothesis to get:

$$v_1 :: (\text{rev}(\text{rev}(b)) <> \text{rev}(\text{rev}(a))) = \text{rev}(\text{rev}(b)) <> v_1 :: \text{rev}(\text{rev}(a))$$

which generalises to the following lemma:

$$\forall X : T,\ \forall U, V : \text{list}(T).\ X :: (U <> V) = U <> X :: V \tag{26}$$

If an induction proof is able to establish this conjectured formula, the following wave-rule would be available:

$$U <> \boxed{X :: \underline{V}}^{\uparrow} \Rightarrow \boxed{X :: \underline{U <> V}}^{\uparrow}$$

Note how this wave-rule would allow further rewriting and completing the proof. As the reader may now notice, (26) is not a valid lemma. But even if it is not valid we can still exploit the information that it provides. If we look carefully at it, we will notice that the wave-front term, i.e. $X :: \ldots$, introduced by the step case proof obligation has to move outwards past both $<>$ and U. This observation enables to deduce that correcting (20) can be achieved by performing one of the following actions:

- Emptying one of the lochs, i.e. to force $A = nil$ or $B = nil$.
- Eliminating the dyke, i.e. to force $A = B$.

From the above actions, we prefer the latter. This strategy has been implemented by switching the positions of these variables in one side of the expression, looking for a pattern of the form:

$$F1(A, F2(X, B)) = F2(X, F1(A, B))$$

or any possible combination, e.g. $F2(X, F1(A, B)) = F1(A, F2(X, B))$.

The critic definition depicted in Fig. 4 shows the general mechanism.

CRITIC wave

Input:	$Plan$,	; Current plan, node,
	node($Plan$, [], $Goal$),	; and sequent.
	current_node($Plan$, $Address$, $H \vdash G$)	
Preconditions:	disprove($H \vdash G$),	; Current goal is faulty.
	lemma_calculation_applicable($H \vdash G$, $BlockedSide$, $RippledSide$),	
	calculated_lemma($H \vdash G$, $BlockedSide$, $Lemma$),	
	lochs_dykes($Lemma$, X, Y),	; The suggested lemma
	switch(X, Y, $Goal$, $Side$, $NewGoal$),	; follows the lochs and
	not disprove([] $\vdash NewGoal$)	; dykes patterns.
Patch:	resume_plan($Plan$, [], [] $\vdash NewGoal$)	

Meanings of the meta-logical terms:

- disprove(Seq) means that it is possible to find a counter-example for Seq.
- lemma_calculation_applicable(Seq, B, R) means that lemma calculation technique is applicable. B and R are the blocked and the rippled side, respectively.
- calculated_lemma(Seq, B, $Lemma$) calculates $Lemma$ to complete a proof.
- lochs_dykes($Lemma$, X, Y) means that $Lemma$ matches the lochs and dykes patterns and that X and Y are in wrong position within the term structure.
- switch(Exp_1, Exp_2, F, $Side$, $NewF$) is used to exchange the positions of subexpressions Exp_1 and Exp_2 in side $Side$ of F.

Fig. 4. Exploiting lochs and dykes

4 Implementation Aspects

Correcting faulty conjectures by adding conditions gives rise to the problem of finding a proof for conditional equations. Generally speaking, we now have goals of the form:

$$C[N] \rightarrow P[N] \vdash C[\boxed{S(\underline{N})}] \rightarrow P[\boxed{S(\underline{N})}]$$

where C, P, and S are terms with a distinguished argument, C is the antecedent, and S any constructor function.

These kinds of goals introduce technical problems in proofs by induction. This is because the antecedents get in the way in an actual proof. We have extended the capabilities of the proof planner to cope in such situations. We use two different strategies. In the first one, we allow fertilization once we have proved that the condition of the induction hypothesis holds, we called this *conditional weak fertilization.* In the second one, we split a proof into cases using the condition of the induction hypothesis and its negation. These strategies have also been implemented as proof critics, thus preserving the core of the system.

5 Comparison to Related Work

5.1 Resolution-Based Abduction Mechanisms

As we have previously mentioned, most of the mechanisms for driving the generation of abductive hypotheses are based on resolution. [15, 14, 5, 6] have independently proposed a mechanism which, roughly speaking, works as follows:

1. Convert the set of axioms and the given conjecture into clausal form;
2. Perform, let us say, SLD-resolution, using the set of support strategy. If the conjecture is false, this deduction process either does not terminate or results in a finitely failed AND/OR proof search tree with the leaves labelled as unprovable goals. If sufficient of these goals were true then the conjecture would be provable.
3. Perform an analysis on the unprovable goals and build from it (often more than) one condition that logically implies the goal.

As the reader may now suspect, the search space generated by this procedure normally is huge, and so is the number of dead end goals. The latter fact is of much importance, since it causes a combinatorial explosion in the process of building a consistent, nontrivial, and minimal condition. On the other hand, our mechanism avoids the combinatorial explosion because the proof plans technique carefully guides the search for an inductive proof and assists the detection, isolation, and analysis of faults.

5.2 PreS

In [7], the authors propose a technique, they call PreS, to correct faulty conjectures. PreS works as a separate module of an inductive theorem prover. When given faulty conjecture G, PreS is aimed at synthesising P such that $P \rightarrow G$ holds. P's definition is built according to the success or failure at establishing base and step cases of inductive proofs. We illustrate this by example.

Consider again non-theorem (1). From Sect. 3.2, we know that a proof attempt, using two step induction, results in

- success in the first base case ($N = 0$);
- failure, in the second base case ($N = \mathrm{s}(0)$); and
- success in the step case if we take $\mathrm{double}(\mathrm{half}(n)) = n$ as the induction hypothesis, and $\mathrm{double}(\mathrm{half}(\mathrm{s}(\mathrm{s}(n)))) = \mathrm{s}(\mathrm{s}(n))$ as the conclusion.

PreS records the following observations:

$$P(0) \text{ is true}, \quad P(\mathrm{s}(0)) \text{ is false}, \quad P(\mathrm{s}(\mathrm{s}(N))) \text{ if } P(N)$$

which actually is the recursive definition of the predicate even.

This approach is interesting in that the definition of P is built using synthesis techniques of the proofs as programs paradigm. Regrettably, PreS is explained only by example, no general mechanism is defined, and no characterisation of failure is provided. For instance, it is not clear how PreS manages, if it does, faulty conjectures in which base cases go through and nested inductions (with possibly generalisations) are required to complete a proof; which is a common situation when proving properties about lists or trees. Our mechanism, on the other hand, captures the restricted way in which the proof of a conjecture that exhibits a particular kind of fault can fail, and provides a general mechanism to patch such failures.

6 Results and Further Work

The strategies presented in this paper have been built upon CLAM v3.1 [17] as a collection of critics. CLAM v3.1 was especially designed to realise the proof critics technique, see [9], described in Sect. 2.2.

We tested our mechanism by making it correct a set of 45 faulty conjectures that included the sorts of faults mentioned in Sect. 3. It proved to be capable of correcting 80% of them. It corrected 72.3% of false conjectures with wrong definitions in boundary values; 72.3% of faulty conjectures with wrong definitions beyond boundary values; and 91.67% of non-theorems in which the fault consisted of wrong definitions in the properties of operators. Table 1 shows some example non-theorems that were successfully corrected.

Our approach only finds one among several possible corrections to a non-theorem. Such a correction is

- consistent if a successful proof plan is found; and

- non-trivial because it consists of either the negation of case conditions provided by a well-founded rule of inference (mathematical induction), or well-defined predicates.

Minimality, however, requires a non-trivial subsumption checking algorithm. We are currently working on this.

Table 1. Example non-theorems successfully corrected. The predicate oddl returns true whenever its input, a list of objects, is of length odd. qrev is the tail reverse function. x, y, a, b, and c in this table represent universally quantified variables and range over either the Peano natural numbers or lists

Critic	Non-Theorems	Theorems
Fig. 1	$\text{length}(a <> b) > \text{length}(a)$	$b \neq \text{nil} \rightarrow \text{length}(a <> b) > \text{length}(a)$
	$\text{length}(a) < \text{length}(a <> b)$	$b \neq \text{nil} \rightarrow \text{length}(a) < \text{length}(a <> b)$
	$\text{half}(x) < \text{double}(x)$	$x \neq 0 \rightarrow \text{half}(x) < \text{double}(x)$
	$\text{half}(x) < x$	$x \neq 0 \rightarrow \text{half}(x) < x$
	$x < \text{double}(x)$	$x \neq 0 \rightarrow x < \text{double}(x)$
	$x + y > x$	$y \neq 0 \rightarrow x + y > x$
	$x + y > \text{s}(x)$	$y > \text{s}(0) \rightarrow x + y > \text{s}(x)$
Fig. 1 and Fig. 3	$\neg\text{even}(x)$	$\text{odd}(x) \rightarrow \neg\text{even}(x)$
	$\neg\text{odd}(x)$	$\text{even}(x) \rightarrow \neg\text{odd}(x)$
	$\text{double}(\text{half}(x)) = x$	$\text{even}(x) \rightarrow \text{double}(\text{half}(x)) = x$
	$\text{double}(\text{half}(x)) \neq x$	$\text{odd}(x) \rightarrow \text{double}(\text{half}(x)) \neq x$
	$\text{even}(x) \rightarrow \text{even}(x + y)$	$\text{even}(y) \rightarrow (\text{even}(x) \rightarrow \text{even}(x + y))$
	$\neg\text{even}(\text{length}(a))$	$\text{oddl}(a) \rightarrow \neg\text{even}(\text{length}(a))$
	$\text{odd}(\text{length}(a))$	$\text{oddl}(a) \rightarrow \text{odd}(\text{length}(a))$
Fig. 4	$a <> (b <> c) = (a <> c) <> b$	$a <> (c <> b) = (a <> c) <> b$
	$\text{rev}(\text{rev}(a <> b)) = b <> a$	$\text{rev}(\text{rev}(b <> a)) = b <> a$
	$\text{rev}(a <> b) = \text{rev}(a) <> \text{rev}(b)$	$\text{rev}(a <> b) = \text{rev}(b) <> \text{rev}(a)$
	$a <> \text{rev}(b) = \text{qrev}(b, a)$	$\text{rev}(b) <> a = \text{qrev}(b, a)$
	$a <> b = b <> a$	$b <> a = b <> a$
	$\text{rev}(a <> x :: \text{nil}) = \text{rev}(a) <> x :: \text{nil}$	$\text{rev}(a <> x :: \text{nil}) = x :: \text{nil} <> \text{rev}(a)$

OYSTER has been especially designed to be applied in the problem of computer program synthesis. We would like to apply the strategies outlined in this paper in the correction of faulty computer program specifications. This process may involve the creation of guards to constrain the input domain of the synthesised code. Note the similarity between these guards and the conditions that transform non-theorems into theorems.

References

1. Bundy, A.: The Use of Explicit Plans to Guide Inductive Proofs. In 9th Conference on Automated Deduction. Lusk, R. and Overbeek, R.(Eds.). (1988) 111–120. Longer version available from Edinburgh as DAI Research Paper No. 349.
2. Bundy, A. and van Harmelen, F. and Hesketh, J. and Smaill, A.: Experiments with Proof Plans for Induction. Journal of Automated Reasoning **7** (1991) 303–324.
3. Bundy, A. and van Harmelen, F. and Horn, C. and Smaill, A.: The Oyster-Clam system. In Proceedings of the 10th International Conference on Automated Deduction. Springer-Verlag. Stickel, M.E. (Ed.). (1990) 647–648.
4. Bundy, A. and Stevens, A. and van Harmelen, F. and Ireland, A. and Smaill, A.: Rippling: A Heuristic for Guiding Inductive Proofs. Artificial Intelligence **62** (1993) 182–253.
5. Cox, P.T. and Pietrzykowski, T.: Causes for Events: Their Computation and Applications. Lecture Notes in Computer Science: Proceedings of the 8th International Conference on Automated Deduction. Siekmann, J. (Ed.) Springer-Verlag. (1986) 608–621.
6. Finger, J.J.:RESIDUE: A deductive approach to design synthesis. Research Report STAN-CS-85-1035. Stanford University (1985).
7. Franova, M. and Kodratoff, Y.: Predicate Synthesis from Formal Specifications. Proceedings of ECAI-92. (1992) 87–91.
8. Gordon, M.J. and Milner, A.J. and Wadsworth, C.P.: Edinburgh LCF - A mechanised logic of computation. Lecture Notes in Computer Science **78** (1979).
9. Ireland, A.: The Use of Planning Critics in Mechanizing Inductive Proofs. International Conference on Logic Programming and Automated Reasoning – LPAR 92, St. Petersburg. Lecture Notes in Artificial Intelligence **624**. Voronkov A. (Ed.). Springer-Verlag. (1992) 178–189.
10. Ireland, A. and Bundy, A.: Using Failure to Guide Inductive Proof. Technical Report, Department of Artificial Intelligence (1992). Available from Edinburgh as DAI Research Paper 613.
11. Monroy, R.: Abduction Mechanisms. Working Paper No. 254, Department of Artificial Intelligence, Edinburgh Univerisity (1994).
12. Paul, G.: Approaches to Abductive Reasoning: an Overview. Artificial Intelligence Review, vol. 7, 109–152. Kluwer Academic Publisher (1993).
13. Peirce, C.S.: Collected papers of Charles Sanders Peirce. Vol. 2, 193. Harston, C. and Weiss, P. (Eds.) Harvard University Press. (1959).
14. Poole, G. and Goebel, R. and Aleliunas, R.: Theorist: a logical reasoning system for defaults and diagnosis. The Knowledge Frontier: Essays in Representation of Knowledge. Cercone, N. and McCalla, G. (Eds.), Springer-Verlag (1987) 331–352.
15. Pople, H.E.: On the Mechanization of Abductive Logic. Proceedings of the third IJCAI. Nilsson, N. (Ed.). (1972) 147–152.
16. Selman, B. and Levesque, H.L.: Abductive and Default Reasoning: A Computational Core. In Proccedings of the 8th National Conference on Artificial Intelligence. (1989) 343–348.
17. van Harmelen, F.: The CLAM Proof Planner, User Manual and Programmer Manual. Technical Paper 4. Department of Artificial Intelligence, Edinburgh University. 1989.